

Aberystwyth University

Data mining the yeast genome in a lazy functional language

Clare, Amanda; King, Ross Donald

Publication date:
2003

Citation for published version (APA):

Clare, A., & King, R. D. (2003). *Data mining the yeast genome in a lazy functional language*.
<http://hdl.handle.net/2160/130>

General rights

Copyright and moral rights for the publications made accessible in the Aberystwyth Research Portal (the Institutional Repository) are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the Aberystwyth Research Portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the Aberystwyth Research Portal

Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

tel: +44 1970 62 2400
email: is@aber.ac.uk

Data mining the yeast genome in a lazy functional language

Amanda Clare and Ross D. King

Computational Biology Group, Department of Computer Science,
University of Wales Aberystwyth, Penglais, Aberystwyth, SY23 3DB, UK
Tel: +44-1970-622424 Fax: +44-1970-628536
`rdk@aber.ac.uk`

Abstract. Critics of lazy functional languages contend that the languages are only suitable for toy problems and are not used for real systems. We present an application (PolyFARM) for distributed data mining in relational bioinformatics data, written in the lazy functional language Haskell. We describe the problem we wished to solve, the reasons we chose Haskell and relate our experiences. Laziness did cause many problems in controlling heap space usage, but these were solved by a variety of methods. The many advantages of writing software in Haskell outweighed these problems. These included clear expression of algorithms, good support for data structures, abstraction, modularity and generalisation leading to fast prototyping and code reuse, parsing tools, profiling tools, language features such as strong typing and referential transparency, and the support of an enthusiastic Haskell community. PolyFARM is currently in use mining data from the *Saccharomyces cerevisiae* genome and is freely available for non-commercial use at <http://www.aber.ac.uk/compsci/Research/bio/dss/polyfarm/>.

1 Declarative languages in data mining

Data mining may at first seem an unusual area for applications for declarative languages. The requirements for data mining software used to be simply speed and ability to cope with large volumes of data. This meant that most data mining applications have been written in languages such as C and C++*. Java has also become very popular recently with the success of the Weka toolkit [1]. However, as the volumes of data grow larger, we wish to be more selective and mine only the interesting information, and so emphasis begins to fall on complex data structures and algorithms in order to achieve good results [2].

The declarative community have in fact been involved in data mining right from the start with Inductive Logic Programming (ILP) [3, 4]. ILP uses the language of logic (and usually Prolog or Datalog) to express relationships in the data. Machine learning is used to induce generalisations of these relationships,

* A selection of current data mining software can be found at <http://www.kdnuggets.com/software/>

which can be later applied to future data to make classifications. The advantage of ILP is that complex results can be extracted, but as the results are expressed symbolically (in logic) the results are still more intelligible and informative than traditional numerical learners such as neural networks.

Some of the earliest successful applications of ILP were to computational biology. Perhaps the best known example was the mutagenesis problem [5]: the task of learning whether a chemical is mutagenic or not, given the atoms, bonds and structures within its molecules. Computational biology is a new and exciting field. Recent advances in DNA sequencing, microarray technology and other large-scale biological analysis techniques are now producing vast databases of information. These databases await detailed analysis by biologists. Due to the amount of data, automatic techniques such as data mining will be required for this task. The explosion in computational biology data will revolutionise biology, and new algorithms and solutions are needed to process this data.

For our work in computational biology, we needed to develop a data mining algorithm that would find frequent patterns in large amounts of relational data. Our data concerns the 6000 genes in the yeast genome (*Saccharomyces cerevisiae*) and our aim is to use these patterns as the first stage in learning about the biological functions of the genes. The data is both structured and relational. We also wanted a solution that was capable of running in a distributed fashion on a Beowulf cluster to make best use of the hardware resources we have available.

In the following sections of this paper we describe the problem we wish to solve, the solution we produced, the reasons we chose to use Haskell for the implementation and the problems and advantages we had using Haskell.

2 The requirements

2.1 The data

There are more than 6000 potential genes of the yeast *S. cerevisiae*. The yeast genome was sequenced in 1996 [6] and has been well studied as a model organism both before and after it was sequenced. However, despite its relatively small size and intensive study, the biological function of 30% of its genes is still unknown. We would like to apply machine learning to learn the relationship between properties of yeast genes and their biological functions, and hence make predictions for the functions of the genes that currently have no known function.

For each gene we collect as much information as possible from public data sources on the Internet. This includes data that is relational in nature, such as predicted secondary structure and homologous proteins.

Predicted secondary structure is useful because the shape and structure of a gene's product can give clues to its function. Protein structure can be described at various levels. The primary structure is the amino acid sequence itself. The secondary structure and tertiary structure describe how the backbone of the protein is arranged in 3-dimensional space. The backbone of the protein makes hydrogen bonds with itself, causing it to fold up into arrangements known as

alpha helices, beta sheets and random coils. Alpha helices are formed when the backbone twists into right-handed helices. Beta sheets are formed when the backbone folds back on itself to make pleats. Random coils are neither random, nor coils, but are connecting loops that join together the alpha and beta regions. The alpha, beta and coil components are what is known as secondary structure. The secondary structures then fold up to give a tertiary structure to the protein. This makes the protein compact and globular. Our data is composed of predicted secondary structure information, which has a sequential aspect - for example, a gene might begin with a short alpha helix, followed by a long beta sheet and then another alpha helix. This spatial relationship between the components is important.

Data about homologous proteins is also informative. Homologous proteins are proteins that have evolved from the same ancestor at some point in time, and usually still share large percentages of their DNA composition. We can search publicly available databases of known proteins to find such proteins that have sequences similar to our yeast genes. These proteins are likely to be homologous, and to share common functions, and so information about these is valuable. For example, if we knew that the yeast genes that have homologs very rich in the amino acid lysine tend to be involved in ribosomal work, then we could predict that any of the genes of unknown function that have homologs rich in lysine could also be producing ribosomal proteins.

Figures 1 and 2 demonstrate the contents of our databases.

The overall method we use is the same as the method used in our work on predicting the functions of genes in the *M. tuberculosis* and *E. coli* genomes [7]. For this we use association mining to discover frequent patterns in the data. Then we use these patterns as attributes into a machine learning algorithm in order to predict gene function. The association mining stage is the concern of this paper.

2.2 Association rule mining

Association rule mining is a common data mining technique that can be used to produce interesting patterns or rules. Association rule mining programs count frequent patterns (or “associations”) in large databases, reporting all that fall above a minimum frequency threshold known as the “support”. The standard example used to describe this problem is that of analysing supermarket basket data, to see which products are frequently bought together. Such an association might be “minced beef and pasta are bought by 30% of customers”. An association rule might be “**if** a customer buys minced beef and pasta **then** they are 75% likely to also buy spaghetti sauce”.

The amount of time taken to count associations in large databases has led to many clever algorithms for counting, and investigations into aspects such as minimising candidate associations to count, minimising IO operations to read the database, minimising memory requirements and parallelising the algorithms. Certain properties of associations are useful when minimising the search space. Frequency of associations is **monotonic**: if an association is not frequent, then

```

orf(yor034c).

hom(yor034c,p31947,b1.0e-8.4.0e-4).
sq_len(p31947,b16_344).
mol_wt(p31947,b1485_38502).
classification(p31947,homo).
db_ref(p31947,prints).
db_ref(p31947,embl).
db_ref(p31947,interpro).

hom(yor034c,p29431,b4.5e-2.1.1).
sq_len(p29431,b483_662).
mol_wt(p29431,b53922_74079).
classification(p29431,buchnera).
keyword(p29431,transmembrane).
keyword(p29431,inner_membrane).
db_ref(p29431,pir).
db_ref(p29431,pfam).

hom(yor034c,q28309,b4.5e-2.1.1).
sq_len(q28309,b16_344).
mol_wt(q28309,b1485_38502).
classification(q28309,canis).
keyword(q28309,transmembrane).
db_ref(q28309,prints).
db_ref(q28309,gcldb).
db_ref(q28309,interpro).

hom(yor034c,p14196,b0.0_1.0e-8).
sq_len(p14196,b344_483).
mol_wt(p14196,b38502_53922).
classification(p14196,dictyostelium).
keyword(p14196,repeat).
db_ref(p14196,embl).

```

Fig. 1. A simplified portion of the homology data for the yeast gene YOR034C. Details are shown of four SWISSPROT proteins (p31947, p29431, q28309, p14196) that are homologous to this gene. Each has facts such as sequence length, molecular weight, keywords and database references. The classification is part of a hierarchical taxonomy.

```

orf(yil137c).

ss(yil137c,1,c).
coil_len(1,gte10).
ss(yil137c,2,b).
beta_len(2,gte7).
ss(yil137c,3,c).
coil_len(3,b6_10).
ss(yil137c,4,b).
beta_len(4,gte7).
ss(yil137c,5,c).
coil_len(5,b3_4).
ss(yil137c,6,b).
beta_len(6,b6_7).
ss(yil137c,7,c).
coil_len(7,b3_4).
ss(yil137c,8,b).
beta_len(8,gte7).
ss(yil137c,9,c).
coil_len(9,b6_10).
ss(ytyil137c,10,b).
beta_len(10,b3_4).
alpha_dist(yil137c,b36.2_47.6).
beta_dist(yil137c,b19.1_29.1).
coil_dist(yil137c,b6.2_39.5).
neighbour(1,2,b).
neighbour(2,3,c).
neighbour(3,4,b).
neighbour(4,5,c).
neighbour(5,6,b).
neighbour(6,7,c).
neighbour(7,8,b).
neighbour(8,9,c).
neighbour(9,10,b).

```

Fig. 2. A portion of the structure data for the yeast gene YIL137C. The secondary structure elements (a - alpha, b - beta, c - coil) are numbered sequentially, their lengths are given, and neighbours are made explicit. Overall distributions are also given.

no specialisations of this association are frequent (if *pasta* is not frequent, then $pasta \wedge mince$ cannot be frequent). And if an association is frequent, then all of its parts or subsets are also frequent.

Perhaps the best known association rule algorithm is APRIORI [8]. It works on a levelwise basis, guaranteeing to take at most $d + 1$ passes through the database, where d is the maximum size of a frequent association. First a pass is made through the database where all singleton associations are discovered and counted. All those falling below the minimum support threshold are discarded. The remaining sets are “frontier sets”. Next, another pass through the database is made, this time discovering and counting frequencies of possible 1-extensions that can be made to these frontier sets by adding an item. For example, $\{pasta\}$ could be extended to give $\{pasta, mince\}$, $\{pasta, sauce\}$, $\{pasta, wine\}$. Whilst the basic idea is a brute force search through the space of associations of ever increasing length, APRIORI reduces the amount of associations that have to be counted by an intelligent algorithm (APRIORIGEN) for generation of candidate associations. APRIORI was one of the early association mining algorithms, but its method of generating candidate associations to count is so efficient that it has been popular ever since.

APRIORI, along with most other association rule mining algorithms, applies to data represented in a single table, i.e non-relational data. For relational data we need a different representation.

2.3 First order association mining

When mining relational data we need to extend ordinary association mining to relational associations, expressed in the richer language of first order predicate logic. The associations are existentially quantified conjunctions of literals.

Definition 1. *A term is either a constant or a variable, or an expression of the form $f(t_1, \dots, t_n)$ where f is an n -place function symbol and t_1, \dots, t_n are terms.*

Definition 2. *An atom is an expression of the form $p(t_1, \dots, t_n)$ where p is an n -place predicate symbol and t_1, \dots, t_n are terms.*

Definition 3. *A literal is an atom or the negation of an atom.*

Some examples of associations are:

$$\begin{aligned} \exists X, Y : & \text{buys}(X, \text{pizza}) \wedge \text{friend}(X, Y) \wedge \text{buys}(Y, \text{coke}) \\ \exists X, Y : & \text{gene}(X) \wedge \text{similar}(X, Y) \wedge \text{classification}(Y, \text{virus}) \wedge \text{mol_weight}(Y, \text{heavy}) \end{aligned}$$

Dehaspe and DeRaedt [9] developed the WARMR algorithm for data mining of first order associations. It works in a similar manner to APRIORI, extending associations in a levelwise fashion, but with other appropriate methods for candidate generation, to eliminate counting unnecessary, infrequent or duplicate associations. WARMR also introduces a language bias that allows the user

to specify modes, types and constraints for the predicates that will be used to construct associations and hence restrict the search space. The language bias of a learning algorithm is simply the set of factors which influence hypothesis selection [10]. Language bias is used to restrict and direct the search.

2.4 Distributed association mining

As the size of data to be mined has increased, algorithms have been devised for parallel rule mining, both for machines with distributed memory [11–14] (“shared-nothing” machines), and, more recently, for machines with shared memory [15]. These algorithms have introduced more complex data representations to try to speed up the algorithms, reduce I/O and use less memory. Due to the size and nature of this type of data mining, it is often the case that even just keeping the candidate associations in memory is too much and they need to be swapped out to disk, or recalculated every time on the fly. The number of I/O passes through the database that the algorithm has to make can take a substantial proportion of the running time of the algorithm if the database is large. Parallel rule mining also raises issues about the best ways to partition the work.

This type of rule mining is of interest to us because we have a Beowulf cluster of machines, which can be used to speed up our processing time. This cluster is a network of around 60 shared-nothing machines each with its own processor and between 256M and 1G memory per machine, with one machine acting as scheduler to farm out portions of work to the others.

2.5 Distributed first order association mining

The version of WARMR that was available at the time was unable to handle the quantity of data that we had, so we needed to develop a WARMR-like algorithm that would deal with an arbitrarily large database.

The program should count associations in relational data, progressing in a levelwise fashion, and making use of the parallel capabilities of our Beowulf cluster. We use Datalog* as the language to represent the database. When the database is represented as a flat file of Datalog facts in plain uncompressed text, each gene has on average 150K of data associated with it (not including background knowledge). This is in total approximately 1G for the whole yeast genome when represented in this way. Scaling is a desirable feature of any such algorithm - it should scale up to genomes that are larger than yeast, it should be able to make use of additional processors if they are added to the Beowulf cluster in the future, and indeed should not rely on any particular number of processors being available.

* Datalog [16] is the language of function free and negation free Horn clauses (Prolog without functions) and as a database query language it has been extensively studied. Datalog and SQL are incomparable in terms of expressiveness. Recursive queries are not possible in SQL, and Datalog needs the addition of negation to be more powerful than SQL

The two main options for parallelisation considered by most association mining algorithms are partitioning the associations or partitioning the database.

Partitioning the candidate associations In this case, it is difficult to find a partition of the candidate associations that optimally uses all available nodes of the Beowulf cluster without duplication of work. Many candidates share substantial numbers of literals, and it makes sense to count these common literals only once, rather than repeatedly. Keeping together candidates that share literals makes it difficult to produce a fair split for the Beowulf nodes.

Partitioning the database The database is more amenable to partitioning, since we have more than 6000 genes, each with their own separate data. Division of the database can take advantage of many Beowulf nodes. Data can be partitioned into pieces that are small enough to entirely fit in memory of a node, and these partitions can be farmed out amongst the nodes, with nodes receiving extra partitions of work when they finish. Partitioning the database means that we can use the levelwise algorithm, which requires just d passes through the database to produce associations of length d . In this application we expect the size of the database to be more of an issue than the size of the candidates.

Although a distributed algorithm necessarily will do extra work to communicate the counts, candidates or data between the machines, the investment in this distributed architecture pays off as the number of machines is increased.

3 The solution

The system would serve two purposes:

1. To provide a immediate solution to mine the homology and structure data.
2. To become a platform for future research into incorporating more knowledge of biology and chemistry into this type of data mining (for example, specific biological constraints and hierarchies).

3.1 Associations

The patterns to be discovered are first order associations. An association is a conjunction of literals (actually existentially quantified, but written without the quantifier where it is clear from the context). Examples of associations are:

$$\begin{aligned} & \textit{pizza}(X) \wedge \textit{buys}(\textit{bill}, X) \wedge \textit{likes}(\textit{sam}, X) \\ & \textit{gene}(X) \wedge \textit{similar}(X, Y) \wedge \textit{keyword}(Y, \textit{transmembrane}) \end{aligned}$$

Associations are constructed in a levelwise manner. At each level, new, candidate associations are generated by specialisation of associations from the previous level under θ -subsumption.

Definition 4. *An association a_1 θ -subsumes an association a_2 if and only if there exists a substitution θ such that $a_1\theta \subseteq a_2$.*

This specialisation is achieved by extension of each of the previous associations by each of the literals in the language that are allowed by the language bias. Candidate associations are counted against the database, and pruned away if their support does not meet the minimum support threshold (θ -subsumption is monotonic with respect to frequency). The surviving candidates become the frequent association set for that level and are used to generate the next level. The algorithm can be used to generate all possible frequent associations, or to generate associations up to a certain length (level).

3.2 Farmer, Worker and Merger

The system we developed is called PolyFARM (Poly-machine First-order Association Rule Miner). To the best of our knowledge PolyFARM is the first system to do distributed first order association mining. There are three main parts to the PolyFARM system:

Farmer Reporting of results so far, and candidate association generation for the next level

Worker Candidate frequency counting on a subset of the database

Merger Collation and compaction of Worker results to save filespace

The candidate associations are generated once, centrally, by the Farmer process, using the language bias and the frequent associations from the previous level. The generation process also checks candidates against a list of infrequent associations from the previous level, to ensure that no part of an association is already known to be infrequent.

The database is partitioned and each Worker process reads in all the candidates, its own database partition and the common background knowledge. Candidates are evaluated (counted) against the database partition, and the results are saved to file (the Beowulf has no shared memory, and we do not rely on any PVM-like architectures). When all Workers have completed, the Farmer collects in the files of counts produced by the Workers. It prunes away the infrequent associations (saving them for future reference), and displays the results so far. Then the Farmer generates the next level of candidates, and the cycle begins again.

A single Worker represents counting of a single partition of a database. On the Beowulf cluster, each node will be given a Worker program to run. When the node has completed, and the results have been saved to a file, the node can run another Worker program. In this way, even if there are more partitions of the database than nodes in the Beowulf cluster, all partitions can be counted within the memory available.

In generating a file of counts from each Worker (each database partition), so many files can be generated that filespace could become an issue. So we introduce the third step - Merger. Merger collates together many Worker files

into one single file, saving space. Merger can be run at any time, when filespace needs compacting. Finally, Farmer will simply read in the results from Merger, rather than collating Workers' results itself. A diagram of how the three steps interact is given in Figure 3.

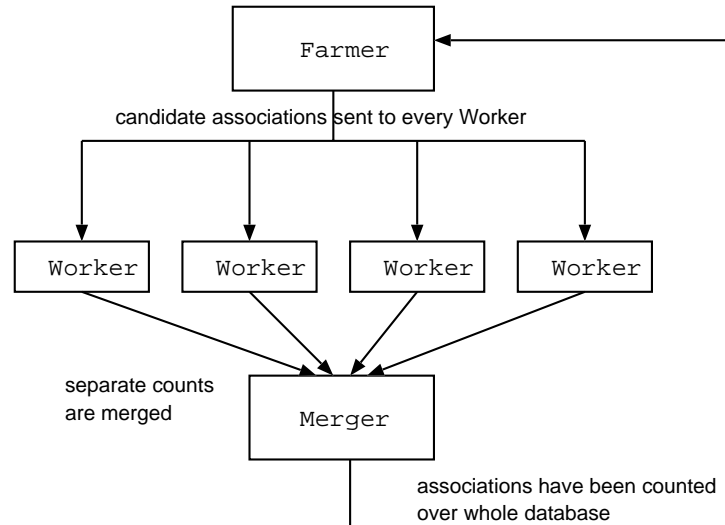


Fig. 3. Farmer, Worker and Merger

This solution addresses 2 aspects of scaling:

- Memory: Partitioning data for the Workers means that no Worker need handle more data than can fit in its main memory, no matter how large the database becomes.
- Filespace: Merger means that the buildup of intermediate results is not a filespace issue.

3.3 Association trees

New candidates are generated by extending associations from the previous level. Any literals from the language can be added, as long as they agree with the modes, types and constraints of the language bias, and the whole association does not contain any part that is known to be infrequent. As each previous association can usually be extended by several literals, this leads naturally to a tree-like structure of associations, where literals are nodes in the tree and children of a node are the possible extensions of the association up to that point. Each level in the tree corresponds to a level in the levelwise algorithm (or the length

of an association). At the root of the tree is a single literal, which all associations must contain.

Allowing common parts of associations to be collected up into a tree structure in this way provides several advantages, and was suggested in Luc Dehapse's PhD thesis [17] (p104) as an improvement that could be made to WARMR. Not only is it a compact way of representing associations, but it means that counting can be done efficiently, since common subparts are counted just once. As the associations are first order, some thought is required to make sure that the various possibilities for variable bindings are consistent within an association.

4 Why Haskell?

Why choose a language such as Haskell for an application such as data mining? Haskell was considered because in previous smaller applications it had proved to be an excellent tool for quickly prototyping ideas. Initial concerns about choosing Haskell as a language for this project were

- Would it be fast enough?
- Would the resource usage be reasonable?
- Would there be enough help and support if problems occurred?

Our application does not need to run in real time, and is not time critical, though a reasonable running speed is required. We have complex data and wish to extract complex information. We are searching through large amounts of genomic information, and algorithm correctness is important, because the results can have biological significance and debugging rare or special cases that show up in large amounts of data is extremely time consuming.

Like many other software projects this required many of the techniques at which Haskell excels:

- Complex algorithms and clarity of code
- Data structures and abstraction
- Parsers and pretty printing
- Modularity and fast prototyping
- Good programming support through language features and on-line help

Whilst all of these are arguably available in any programming language if code is written well, high level declarative languages such as Haskell provide more support for the programmer. Since the application uses data expressed in Datalog and methods such as θ -subsumption, we considered using Prolog as an implementation language. However we were reluctant to give up features such as higher order functions, strong typing and profiling tools, and given the knowledge that our input data would consist of ground terms only, we would not be needing full Prolog to deal with it.

5 Disadvantages of using Haskell

Execution time did not turn out to be a major problem. The run time is adequate for our purposes. Time profiling was used to tune the system, and it became apparent that testing for θ -subsumption accounted for most of the time taken. This is due to both the number of subsumption tests required, and the relatively expensive nature of this test. This time was then substantially alleviated and reduced by restricting the database literals to be tested - firstly to those with the correct predicate symbol, and secondly to those whose constant arguments match exactly with the constants in the literal of the association.

The main disadvantage that was faced was dealing with unwanted laziness. In a data mining environment, where all data needs to be read and counted, and all calculations will be used, laziness provides few advantages, and usually takes up huge amounts of heap space while delaying all computation until the last minute. At every step of the coding process it was found that code was easy to get right, but then difficult to get working in practice without running out of memory.

5.1 Reading data

Much of the memory was being used by lazy reading of data. When data was lazily held in string form, it would consist of repetitions of literals and constants, and occupy far more memory than the compiled form that the program actually used.

If Haskell is to be used in earnest for real-world programs it needs to have to have good methods of reading and writing persistent data. Using the built-in `read` function for large amounts of user data or complex data structures (for example large trees of associations) is not appropriate. Apart from being slow, it is also difficult to force code using `read` to be strict. Using a parser generator tool such as Happy (<http://haskell.cs.yale.edu/happy/>) to generate specialist parsers helps by giving an enormous speed increase, and some memory reduction too. The disadvantages were the difficulty in tracking down the source of type errors in generated code (Happy itself provides little in the way of error checking, and type errors come to light only when using the generated Haskell), and the fact that Happy currently doesn't return anything until the whole input is parsed, and then it returns a tree of thunks. However, in the latest version of Happy, there is now a `--strict` flag to make all the productions strict. We used Happy generated parsers for all user-generated data (knowledge base, settings, and background knowledge).

Malcolm Wallace and Colin Runciman's `Binary` library ([18]) (<http://www.cs.york.ac.uk/fp/nhc98/libs/Binary.html>) was considered. Currently only available for the nhc98 compiler, this provides binary data reading and writing. Data can be read and written to file or to memory, and due to the compression used, the amount of data stored will generally be an order of magnitude smaller than if it were stored as text. This also means an order of magnitude saving in memory costs too, as there are no strings to remain on the heap, because the

data is read in directly. The single disadvantage of Binary at the moment is that it is not Standard Haskell, so is not portable between the compilers, and the only compiler providing support for this is nhc98.

The final solution we chose for reading and writing the counted associations that were to be communicated between Beowulf nodes was a simple strict IO class* providing operations `fromSeq` and `toSeq` that read and wrote data from/to a file handle. Instances of this class were written so that `seq` was applied at every stage, and data was then read and written strictly. This provided a Standard Haskell solution.

5.2 Strictness annotations

Laziness elsewhere in the code was also a problem. Several approaches were used to enforce stricter execution and reduce heap usage. The Haskell language provides support for strictness annotations to data and functions. The `seq` function is provided in Haskell to enforce evaluation. `x 'seq' y` will evaluate `x`, enough to check that `x` is not bottom, then discard the result and return `y`. This means that `x` is guaranteed to be evaluated before `y` is considered. However, `seq` forces evaluation only of the top level construct, so its argument is reduced only to weak head normal form. This was often not enough to remove the laziness. Haskell also allows user-defined data types to contain strictness annotations, specifying that each annotated argument to a data constructor will be evaluated when the constructor is applied. This evaluation will also be to weak head normal form only. Careful use of the combination of strictness annotations on data types and `seq` should be enough to force complete evaluation.

`DeepSeq` is a module that provides the `deepSeq` function, which forces complete evaluation rather than the partial evaluation of `seq`. `DeepSeq` is not part of the standard libraries, but has been requested many times by users, so is easy to access on the Internet.

However, all such ways of enforcing evaluation and strictness are the programmer's responsibility, and in practice, their use is not obvious to the average programmer. `DeepSeq` is not yet automatically derivable, and instances must be written for all our datatypes. Finding the correct places in the code to use `seq` or `deepSeq` after the code has been written is not always obvious, and code can still be lazy if any necessary places are overlooked. `deepSeq` can also be an additional time overhead: if a computation is repeatedly carried out on an ever-growing datastructure, then it may be necessary to re-`deepSeq` the whole structure each iteration, even though most of it will have already been evaluated the previous time. Sometimes it can be difficult to find a place to put the `seq` or `deepSeq` functions and extra code needs to be written.

5.3 CPS

Use of Continuation Passing Style (CPS) was another technique we used to enforce evaluation order. By converting parts of the program into CPS some

* We would like to thank Julian Seward for this solution

control could be gained over the order of evaluation. CPS gives the programmer a handle on the continuation computation, the computation that comes next, and so this computation can be moved over a part of the code that will force execution, such as a conditional, case expression or a pattern match. CPS is a purely functional style of coding, and fits elegantly into Haskell’s character. However, encoding parts of the evaluation order in this manner tends to make the code less readable (and occasionally unintelligible!), which negates one of the main reasons for using Haskell. We look forward to the day when automatic code refactoring tools are available that can analyse laziness and automatically reorganise code in a readable manner.

6 Advantages of using Haskell

6.1 Laziness as an advantage?

The main place in the program where laziness was an advantage was in matching associations to the data (testing for subsumption).

A tree structure of associations makes counting efficient since subparts are not repeatedly counted. For example, the associations

$$\begin{aligned} &gene(X) \wedge similar(X, Y) \wedge mol_weight(Y, heavy) \\ &gene(X) \wedge similar(X, Y) \wedge mol_weight(Y, light) \\ &gene(X) \wedge seq_length(X, long) \\ &gene(X) \wedge seq_length(X, short) \end{aligned}$$

would be stored in the natural tree structure shown in Figure 4.

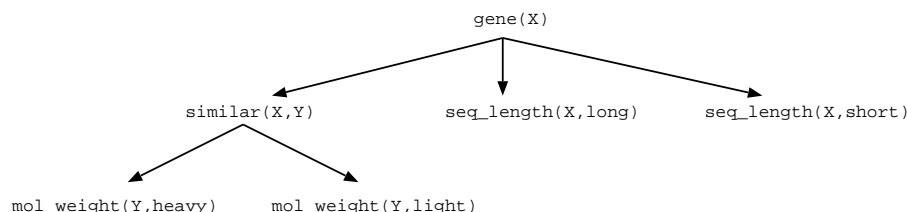


Fig. 4. Associations stored in a tree structure

As we test the associations to see whether or not they match the data for a particular gene in our database, we are aware that individual literals within the association may match the database with more than one possible binding of its variables. The variable bindings must be consistent along a path from root to leaf of the tree. The WARMR system, on which this work is based, was

written in Prolog. In order to efficiently make use of both this tree structure and Prolog’s built-in query subsumption test without repeatedly backtracking and redoing the variable bindings, the authors of WARMR devised the concept of “query packs” [19], requiring an extension to the standard Prolog system. In Haskell, calculating a lazy list of possible variable bindings at each node in the tree ensures that no unnecessary computation is needed. When a complete match is found the alternative possible bindings are no longer required but do not need pruning, since they will never be evaluated due to laziness.

6.2 Algorithm expression

Haskell makes algorithm expression elegant, clear and simple. It simplifies the process of debugging by allowing each function to be tested independently and modified without side effects to other code. Higher order functions and polymorphic types can allow generalisations that make code shorter. We frequently used the predefined higher order functions, such as “map”, “fold”, “filter” and “any”. However, we seldom defined our own, which indicates that we are still not using the full power of Haskell. Pattern matching on datatypes allows complex problems to be broken down into manageable parts. In this code there are algorithms for pruning infrequent associations, generating new extensions to associations given the modes and types supplied by the user for the predicates, testing for θ -subsumption, and various other non-trivial tasks. Haskell is a very clean and concise language, allowing the program to be easily expressed and almost self documenting. We illustrate some of the elegance of Haskell by demonstrating the `makeCombs` function in Figure 5.

`makeCombs` is the function that is responsible for constructing all the possible sets of correct arguments for a new literal that is to be added, given the type constraints and mode constraints for this literal. We work through a list of arguments, each of which has a mode (`Plus`, `Minus`, `PlusMinus` or `ConstList`). Prolog variables will be represented by `Ints`, and constants by `PackedStrings`. We wish to construct all possible combinations of the results, and remember the maximum variable number used in each combination of arguments. Each case can be clearly identified. List comprehensions give a concise way to construct all possible combinations of results without generating many intermediate data structures, and they also allow us to test for constraints such as our type constraints.

6.3 Data structures

The associations are stored in a tree structure to make best use of their common subparts and hence much of the program is based on handling tree-structured data (merging, searching, pruning and path-finding). We also allow hierarchically structured values for the arguments within a literal, and support counts must be propagated through these hierarchies. Haskell provides good support for trees with variable branching factors. In many other parts of the code variable length lists are ideal, since we rarely know in advance how many possible elements

```

makeCombs :: PredID -> [(ArgPosition,Mode)] -> MaxVarNum -> [Pred]
          -> Type -> [(VarCount, [Arg])]

-- When we reach the end of the argument list, just note the
-- maximum variable number used so far
makeCombs predID [] maxVar preds typenums = [(maxVar,[])]

-- if the mode is Plus then this argument must be a variable or constant of
-- the correct type which already exists in the association
makeCombs predID ((n,Plus):otherArgModes) maxVar preds types =
  let otherArgCombs = makeCombs predID otherArgModes maxVar preds types in
  [ (v, (Var i) : alist) | i <- [1..maxVar] ,
    correctVarType i n predID types preds,
    (v, alist) <- otherArgCombs]
  ++ [ (v, a : alist) | a <- correctAtomicTypes n predID types preds,
    (v, alist) <- otherArgCombs]

-- if the mode is Minus then we need to create a new variable, which is
-- numbered 1 greater than the current maximum variable number for this
-- association
makeCombs predID ((n,Minus):otherArgModes) maxVar preds types =
  let otherArgCombs = makeCombs predID otherArgModes maxVar preds types in
  [ (newMaxVar+1, (Var (newMaxVar+1)) : alist) |
    (newMaxVar,alist) <- otherArgCombs]

-- if the mode is PlusMinus, we want both options, Plus and Minus
makeCombs predID ((n,PlusMinus):otherArgModes) maxVar preds types =
  makeCombs predID ((n,Plus):otherArgModes) maxVar preds types
  ++ makeCombs predID ((n,Minus):otherArgModes) maxVar preds types

-- if the argument is to be a constant, we generate all possible constants
makeCombs predID ((n,ConstList cs):otherArgModes) maxVar preds types =
  let otherArgCombs = makeCombs predID otherArgModes maxVar preds types in
  [ (v, (Atomic (packString c)) : alist) | c <- cs ,
    (v,alist) <- otherArgCombs]

```

Fig. 5. The makeCombs function is responsible for constructing all the possible sets of correct arguments for a new literal that is to be added, given type and mode constraints.

will be needed. We had little use for random access arrays since in data mining we usually have to test or count all elements. Lists of lists and other nested structures have been invaluable. We make use of partially indexed lists when the search space can be reduced. Easily definable groupings of data into new data types greatly enhanced code readability and understanding. Haskell has many more useful features that we did not use, such as parameterised data structures. However, we used many of the pre-defined parameterised data structures such as lists and FiniteMap.

6.4 Parsers and pretty printing

Various parsers are needed to read in the knowledge base (a Datalog file), a user settings file describing the language bias and several other parameters, a background knowledge file including hierarchical data, and data which is to be exchanged by nodes of the Beowulf cluster during distributed processing. We also need pretty printing of results.

6.5 Modularity and fast prototyping

Modularity is very important for future experimentation, additions, redesign and modification. Abstraction allowed data types to be changed easily, to give faster execution via a different data structure. Lack of side effects in Haskell means that any changes to code are guaranteed not to interfere with any existing code. When developing this application there were many instances where an algorithm could be achieved via a number of different methods and there were several decisions to be made. The ability to prototype quickly allowed experimentation with research ideas, and this made the project less prone to retention of bad design due to the time invested in writing it, or the time to redesign it.

6.6 Good programming support

In spite of the lack of available books on the language, the wealth of friendly, free and expert advice available from the Haskell community, the various mailing lists and on-line Internet documentation is tremendous. Other good support comes from the tools available, such as time and heap profiling, make-tools, interpreters for easy testing, and features of the language itself such as strong static typing and referential transparency meaning that the compilers can catch most errors.

7 Conclusions

PolyFARM is a data mining application written entirely in Standard Haskell. It is currently in use for analysing computational biology data from the genome of the yeast *S. cerevisiae*. We hope to apply this to the 25,000 genes of the plant genome *A. thaliana* next, and to other genomes in future. This application is not limited to computational biology data, but can be used to mine frequent

associations in any relational data expressed as Datalog. In future we also hope to extend the interface to connect directly to data in standard relational databases, as an additional alternative to using the Datalog format inherited from the field of ILP. We plan to further develop this application for future research in the field of data mining of biological data.

The use of Haskell was a success for this project. A prototype was correctly working within a remarkably short time frame (3 man-weeks), which encouraged us to continue using Haskell. However, we encountered problems with unexpected laziness frequently filling up the heap space. A further 2 months were needed to obtain reasonable resource usage. The heap problems were resolvable with the aid of the excellent profiling tools available, but profiling took time, and solutions required tricks and intuition that are not obvious. Finding all the correct places to use `seq` after the code has been written was difficult. Transforming code into CPS style can make code obscure, and tax the programmer. We would like to see more support for refactoring code to control heap usage.

Haskell was an excellent language for coding the application due to many features, including clear expression of complex algorithms, good support for data structures, abstraction, modularity and generalisation leading to fast prototyping and code reuse, parsing tools, profiling tools, language features such as strong typing and referential transparency which allowed for easy modification of code, and the helpful support of an enthusiastic Haskell community.

PolyFARM is freely available for non-commercial use at <http://www.aber.ac.uk/compsci/Research/bio/dss/polyfarm/>.

References

1. Witten, I.H., Frank, E.: Data Mining: Practical machine learning tools with Java implementations. Morgan Kaufmann, San Francisco (1999)
2. Mannila, H.: Methods and problems in data mining. In: International Conference on Database Theory. (1997)
3. Muggleton, S., ed.: Inductive Logic Programming. Academic Press (1992)
4. Wrobel, S., Džeroski, S.: The ILP description learning problem: Towards a general model-level definition of data mining in ILP. In: FGML-95 Annual Workshop of the GI Special Interest Group Machine Learning (GI FG 1.1.3). (1995)
5. King, R., Muggleton, S., Srinivasen, A., Sternberg, M.: Structure-activity relationships derived by machine learning: The use of atoms and their bond connectives to predict mutagenicity by inductive logic programming. *Proc. Nat. Acad. Sci. USA* **93** (1996) 438–442
6. Goffeau, A., Barrell, B., Bussey, H., Davis, R., Dujon, B., Feldmann, H., Galibert, F., Hoheisel, J., Jacq, C., Johnston, M., Louis, E., Mewes, H., Murakami, Y., Philippsen, P., Tettelin, H., Oliver, S.: Life with 6000 genes. *Science* **274** (1996) 563–7
7. King, R., Karwath, A., Clare, A., Dehaspe, L.: Genome scale prediction of protein functional class from sequence using data mining. In: KDD 2000. (2000)
8. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: 20th International Conference on Very Large Databases (VLDB 94). (1994) Expanded version: IBM Research Report RJ9839, June 1994.

9. Dehaspe, L., De Raedt, L.: Mining association rules in multiple relations. In: 7th International Workshop on Inductive Logic Programming. (1997)
10. Utgoff, P.: Shift of bias for inductive concept learning. In Michalski, R., Carbonell, J., Mitchell, T., eds.: Machine Learning: An Artificial Intelligence Approach, Volume II. Morgan Kaufmann (1986)
11. Park, J.S., Chen, M., Yu, P.: Efficient parallel data mining for association rules. In: CIKM '95. (1995)
12. Agrawal, R., Shafer, J.: Parallel mining of association rules. IEEE Trans. on Knowledge and Data Engineering **8(6)** (1996) 962–969
13. Cheung, D., Ng, V., Fu, A., Fu, Y.: Efficient mining of association rules in distributed databases. IEEE Trans. on Knowledge and Data Engineering **8(6)** (1996) 911–922
14. Han, E., Karypis, G., Kumar, V.: Scalable parallel data mining for association rules. In: SIGMOD '97. (1997)
15. Parthasarathy, S., Zaki, M., Ogihara, M., Li, W.: Parallel data mining for association rules on shared-memory systems. Knowledge and Information Systems **3(1)** (2001) 1–29
16. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Vol. 1 and 2. Computer Science Press, Rockville, Md. (1988)
17. Dehaspe, L.: Frequent Pattern Discovery in First Order Logic. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven (1998)
18. Wallace, M., Runciman, C.: The bits between the lambdas: Binary data in a lazy functional language. In: Proceedings of the International Symposium on Memory Management. (1998)
19. Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J., Vandecasteele, H.: Improving the efficiency of Inductive Logic Programming through the use of query packs. Journal of Artificial Intelligence Research **16** (2002) 135–166